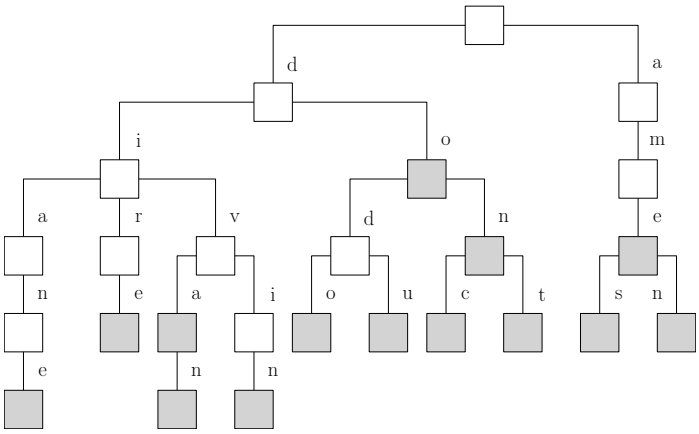
Anagrammes

1 Structure de trie

Considérons l'ensemble de mots suivant :

```
let mots = ["diane"; "dire"; "diva"; "divan"; "divin"; "do"; "dodo";
            "dodu"; "don"; "donc"; "dont"; "ame"; "ames"; "amen"]
```

On peut représenter cet ensemble sous forme d'un arbre d'arité variable, où un nœud grisé signifie que le mot correspondant (c'est-à-dire le mot qu'on lit en allant de la racine au nœud) appartient au dictionnaire : on parle de *trie* pour cette structure de données.

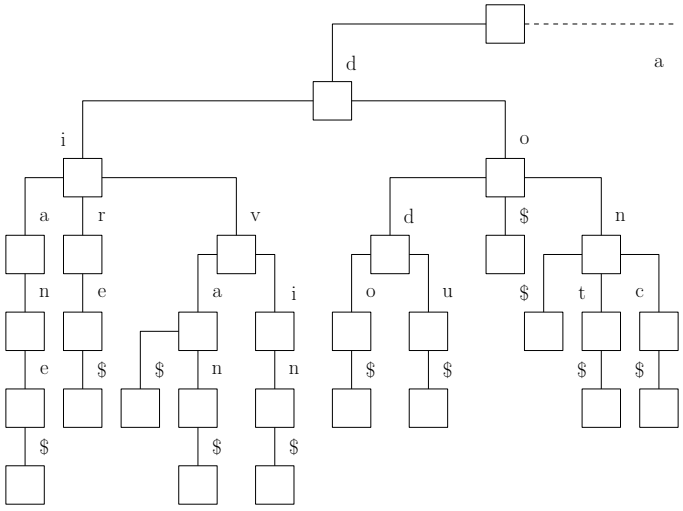


Représentation arborescente de `mots`.

Pour simplifier la programmation, nous allons utiliser une technique assez courante :

- on choisit un caractère qui n'apparaît dans aucun mot de notre dictionnaire (pour nous, ce sera `$`) ;
- ce caractère devient un « marqueur de fin de mot » : il est ajouté à la fin de tous les mots.

Il n'est alors plus nécessaire de distinguer les nœuds correspondant à un mot du dictionnaire et les autres : les mots du dictionnaire sont exactement ceux qui correspondent aux feuilles de notre arbre.



Partie gauche de l'arbre en figure **??**, avec marqueurs de fin de mot.

Ensuite, nous avons déjà parlé de plusieurs représentations mémoire possibles pour un arbre d'arité quelconque :

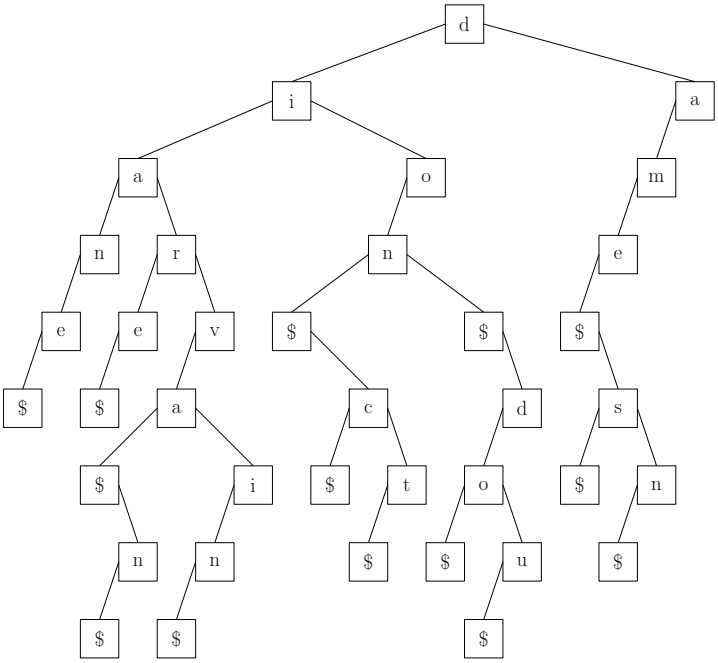
- chaque nœud peut contenir un tableau d'enfants (ici, l'enfant correspondant au caractère `c` pourrait être placé dans la case `int_of_char c` du tableau) ;
- chaque nœud peut aussi contenir une liste d'enfants, ou plutôt une liste de couples (caractère, enfant) (puisque les arêtes portent des étiquettes) ;

— enfin, il est possible de « binariser » l'arbre.

C'est cette solution que nous allons choisir, et l'on définit donc le type suivant :

```
type dict =
  | V
  | N of char * dict * dict
```

On obtient alors (en omettant les nœuds `V`) :



Version binaire de l'arbre de la figure ?.

On impose deux contraintes sur nos arbres :

— un nœud étiqueté par `$` a forcément `V` comme fils gauche ;

— un nœud étiqueté par un caractère autre que `$` n'a jamais `V` comme fils gauche.

1.1 Exercice

Définir une fonction `est_bien_forme` qui vérifie si un `dict` est bien formé.

```
est_bien_forme : dict -> bool
```

À partir de maintenant, les tries passés en argument seront systématiquement supposés bien formés, et toute fonction renvoyant un trie devra obligatoirement renvoyer un trie bien formé.

Un mot (une suite finie de caractères) sera dit *bien formé* s'il contient exactement un caractère `$`, placé en dernière position. Un mot bien formé est donc nécessairement non vide.

1.2 Exercice

Donner une définition (inductive) de la fonction φ qui à un arbre binaire (bien formé) du type ci-dessus associe un ensemble de mots bien formés. Dans l'exemple ci-dessus, on a :

$$\varphi(t) = \{\texttt{"diane"}, \text{"}dire\text{"}, \texttt{"diva"}, \text{"}divan\text{"}, \texttt{"divin"}, \text{"}don\text{""}donc\text{"},$$
$$\texttt{"dont"}, \text{"}dodo\text{"}, \texttt{"dodu"}, \text{"}ame\text{"}, \texttt{"ames"}, \text{"}amen\text{"}\}$$

2 Fonctions utilitaires

On définit l'alias de type suivant :

```
type mot = char list
```

2.1 Exercice

1. Écrire une fonction `mot_of_string` prenant en entrée une chaîne de caractères et renvoyant la liste de ses caractères, avec un caractère `$` rajouté à la fin.

```
mot_of_string : string -> mot
```

```
# mot_of_string "bonjour";;
- : char list = ['b'; 'o'; 'n'; 'j'; 'o'; 'u'; 'r'; '$']
```

2. Écrire une fonction `afficher` qui prend en entrée une liste de caractères et affiche le mot correspondant, suivi d'un retour à la ligne. Les éventuels caractères `$` seront ignorés.

```
afficher : mot -> unit
```

```
# afficher_liste ['b'; 'o'; 'n'; 'j'; 'o'; 'u'; 'r'; '$'];;
bonjour
- : unit = ()
```

Un objet de type mot sera dit bien formé s'il contient exactement un caractère `$`, placé en dernière position.

3 Opérations élémentaires sur les tries

3.1 Exercice

1. Écrire une fonction `cardinal` renvoyant le nombre de mots bien formés contenus dans un dictionnaire.

```
cardinal : dict -> int
```

2. Écrire une fonction `appartient` qui détermine si un certain mot bien formé appartient à un dictionnaire.

```
appartient : dict -> mot -> bool
```

3.2 Exercice

1. Écrire une fonction `ajouter` qui ajoute un mot, supposé bien formé, à un dictionnaire.
2. Écrire une fonction `dict_of_list` prenant en entrée une liste de chaînes de caractères et renvoyant un dictionnaire contenant exactement les mots de cette liste (auxquels on a ajouté un `$`).

```
ajouter : dict -> mot -> dict
dict_of_list : string list -> dict
```

3.3 Exercice

1. Écrire une fonction `afficher_mots` qui affiche tous les mots bien formés appartenant à un dictionnaire (sans les `$` finals), à raison d'un mot par ligne.
2. Écrire une fonction `longueur_maximale` qui renvoie la longueur maximale d'un mot bien formé du dictionnaire (on ne comptera pas le `$` final, et l'on renverra -1 s'il n'y a pas de mot bien formé).
3. Écrire une fonction `afficher_mots_longs` qui affiche tous les mots de longueur supérieure ou égale à l'entier passé en argument (toujours sans compter le `$`).

```
afficher_mots : dict -> unit
longueur_maximale : dict -> int
afficher_mots_longs : dict -> int -> unit
```

4 Lecture de fichier

Vous trouverez quelques fichiers contenant des mots, dont le format est très simple : chaque mot est sur une ligne. Ces mots ne contiennent que des lettres minuscules de a à z, sans signe diacritique (autrement dit, les accents, cédilles et autres ont été retirés des fichiers en français).

— Le fichier `ab.txt` contient 9940 mots anglais, commençant tous par a ou par b.
— Le fichier `10000.txt` contient les 10000 mots anglais les plus courants.
— Le fichier `nettoye.txt` contient 336531 mots français, débarassés de leurs signes diacritiques.

Dans tous les cas, il y a quelques doublons (dûs à la suppression des signes diacritiques).

4.1 Exercice

Écrire une fonction `lire_fichier` qui prend en entrée un nom de fichier (ou plutôt un chemin relatif vers un fichier) et renvoie le dictionnaire correspondant. On supposera que le format est celui décrit ci-dessus (un mot par ligne).

```
lire_fichier : string -> dict
```

```
# cardinal (lire_fichier "ab.txt");;
- : int = 9938
# cardinal (lire_fichier "10000.txt");;
- : int = 9989
# cardinal (lire_fichier "nettoye.txt");;
- : int = 323422
```

5 Filtrage

5.1 Exercice

1. Écrire une fonction `calculer_occurrences` qui prend en entrée une chaîne de caractères `s` et renvoie un `int array` de longueur 256 tel que `t.(i)` soit égal au nombre d'occurrences de `int_of_char i` dans `s`.
2. Écrire une fonction `afficher_mots_contenus` qui prend en entrée un mot sous forme de chaîne de caractères (sans `$` final) et un dictionnaire, et affiche tous les mots du dictionnaire que l'on peut former en utilisant tout ou partie des lettres du mot fourni (en tenant compte des répétitions).
3. Écrire une fonction `afficher_anagrammes` qui prend en entrée un mot sous forme de chaîne de caractères et affiche toutes ses anagrammes présentes dans le dictionnaire. Une *anagramme* d'un mot est un mot constitué exactement des mêmes lettres (avec le même nombre d'occurrences) mais dans un ordre différent (on considérera qu'un mot est anagramme de lui-même).
4. Quel est sont les mots français les plus courts contenant toutes les voyelles ?

```
calculer_occurrences : string -> int array
afficher_mots_contenus : dict -> string -> unit
afficher_anagrammes : dict -> string -> unit
```

5.2 Exercice

1. Écrire une fonction `filtrer_mots_contenus` qui prend les mêmes arguments que `afficher_mots_contenus` mais renvoie un dictionnaire contenant les mots que l'on peut former. On produira directement le dictionnaire, sans commencer par produire la liste des mots.
2. Écrire une fonction `filtrer_mots_contenant` qui fait la même chose que la précédente, sauf qu'on s'intéresse cette fois aux mots *à partir desquels* on peut former le mot fourni (c'est-à-dire ceux contenant toutes les lettres du mot fourni, en tenant compte des répétitions).
3. Écrire une fonction similaire `filtrer_anagrammes`.

6 Décomposition en anagrammes

On appelle *décomposition en anagrammes* d'un mot m dans un dictionnaire d une suite de mots de d qui, mis bout à bout, forment un anagramme de m. Par exemple, `"sans ame"`, `"sa mes na"`, `"a mes ans"` sont des décompositions possibles de `"massena"` avec un dictionnaire français standard.

6.1 Exercice

Dans cet exercice, on s'autorise à générer plusieurs fois une décomposition : par exemple, `"sans ame"` et `"ame sans"` (autrement dit, une décomposition en n mots sera générée $n!$ fois).

1. Écrire une fonction `afficher_decompositions` qui affiche toutes les décompositions en anagrammes d'un mot dans un dictionnaire.
2. Écrire une fonction `decompositions` qui renvoie un dictionnaire contenant toutes ces décompositions. La décomposition `"sans ame"`, par exemple, sera stockée comme un mot de huit caractères (sans compter le `$` final), avec un caractère « espace » entre le « s » et le « a ».

```
afficher_decompositions : dict -> string -> unit
decompositions : dict -> string -> dict
```

6.2 Exercice

Écrire une fonction `decompositions_uniques` qui génère le dictionnaire des décompositions dans lequel deux décompositions ne différant que par l'ordre des mots sont considérées comme identiques (et ne contenant donc que l'une de ces décompositions). On pourra par exemple ne générer que les décompositions pour lesquelles la suite des mots est croissante (dans l'ordre lexicographique).

```
decompositions_uniques : dict -> string -> dict
```